# Solving large zero-one-linear programming problems

H. Crowder, E.L. Johnson, M.W. Padberg

INRIA
TRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports de Recherche

N° 89

# SOLVING LARGE-SCALE ZERO-ONE LINEAR PROGRAMMING PROBLEMS

Harlan CROWDER
Ellis L. JOHNSON
Manfred W. PADBERG

Septembre 1981

RAPPORT DE RECHERCHE

INRIA

Titre : Solving Large-Scale Zero-One Linear Programming Problems

Auteurs : Harlan Crowder

Ellis L. Johnson

Manfred W. Padberg

Résumé :

Dans la littérature de la gestion des affaires et de la recherche opérationnelle, on trouve une multitude prodigieuse de problèmes que l'on peut formuler, en termes mathématiques, à l'aide de variables bivalentes qui doivent satisfaire certaines inégalités ou égalités linéaires et pour lesquelles on veut optimiser une fonction économique linéaire. On soupçonne pourtant -bien qu'elles soient utiles pour l'analyse des problèmes pratiques- que ces formulations ne peuvent pas être résolues par des méthodes exactes. Ce sentiment est fondé en partie sur le fait que le problème de l'optimisation d'une fonction linéaire en variables bivalentes sous des contraintes linéaires appartient à la classe des problèmes NP -complets, c'est-à-dire à la classe des problèmes combinatoires difficiles, pour lesquels on ne connaît pas de bons algorithmes.

Dans cet article nous rendons compte de la solution exacte de dix problèmes de grande taille et de la forme mentionnée. Les données de ces problèmes sont toutes fournies par diverses applications réelles au domaine de la gestion des affaires et elles sont caractérisées par des matrices rationnelles et creuses. Environ la moitié de nos problèmes n'a aucune structure spéciale -sauf d'être creuse ce qui est normal pour les problèmes de grande taille- les autres possèdent des caractéristiques structurelles qui ne sont pas exploitées directement dans notre calcul.

La méthodologie que nous utilisons a produit des résultats numériques impressionnants, surtout pour des problèmes sans caractéristique structurelle.

Les problèmes de notre étude ont été sélectionnés parmi des problèmes de IBM Corporation et de ses clients ; la plupart d'entre eux ont été considérés comme insolubles par des méthodes exactes dans des temps de calcul raisonnables. Les résultats numériques dont nous rendons compte ici réfutent ce sentiment et confirment fortement notre hypothèse selon laquelle on peut optimiser des problèmes de grande taille en variables bivalentes, avec le savoir théorique que l'on a aujourd'hui et avec la technologie existante.

Sur le plan théorique, nos résultats montrent que la méthode des coupes est un instrument indispensable à la résolution exacte de ces problèmes.

Pour arriver à ces conclusions, nous avons conçu et réalisé un logiciel expérimental PIPX qui utilise comme composantes le logiciel MPSX/370 d'IBM pour la programmation linéaire, et le logiciel MIP/370 d'IBM pour le Branch-and-Bound. Notre logiciel PIPX est pourtant automatique, c'est-à-dire qu'il ne nécessite aucune intervention manuelle.

# SOLVING LARGE-SCALE ZERO-ONE LINEAR PROGRAMMING PROBLEMS

Harlan Crowder*, Ellis L. Johnson† and Manfred W. Padberg‡

**ABSTRACT:** The management science and operations research literature abounds with problems which can be appropriately formulated using zero-one variables which have to satisfy certain linear constraints. It is generally assumed, however, that — while useful for the analysis of real-world problem situations — zero-one problems of large-scale are not amenable to solution by exact methods. This is partly so because the general zero-one problem with linear constraints and linear objective function is among the NP-complete or "hard" combinatorial optimization problems for which, to date, no technically good algorithms are available. In this paper we report the *solution to optimality* of ten large-scale zero-one linear programming problems. All problem data come from real-world industrial applications and are characterized by *sparse* constraint matrices with rational data. About half of the sample problems have no apparent special structure; the remainder show structural characteristics which are not exploited directly by our computational procedures. Our methodology produced — by today's standards — impressive computational results, in particular on sparse problems having no apparent special structure. The test problems were given to us from various sources within and outside of the IBM Corporation and, while we have no comparative data on previous solution attempts, most were originally considered not amenable to exact solution in economically feasible computation times. The computational results reported here contradict this sentiment and strongly confirm our hypothesis that a combination of problem preprocessing, cutting planes and clever branch-and-bound techniques permit the optimization of sparse large-scale zero-one linear programming problems, even when no apparent special structure is present, in reasonable computation times. Our results indicate that cutting-planes are an indispensable tool for the exact solution of this class of problem. To arrive at these conclusions, we designed an experimental computer system PIPX which uses the IBM linear programming system MPSX/370 and the IBM integer programming system MIP/370 as building blocks. The entire system is automatic and requires no manual intervention.

**KEYWORDS:** COMPUTATION, ZERO-ONE LINEAR PROGRAMMING, CUTTING PLANES, BRANCH-AND-BOUND, PROBLEM PREPROCESSING, FACETS

---

* IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

† IBM Thomas J Watson Research Center, and Universität Bonn, FRG

‡ New York University, New York, NY 10006, and INRIA, Rocquencourt, France

# Contents

## Tables



## Figures

# 1 Introduction

In this paper we report the *solution to optimality* of ten large-scale zero-one linear programming problems. Two problems in our study have 33 and 40 zero-one variables, respectively; the remaining eight problems have between 200 and 2750 zero-one variables. All problem data come from real-world industrial applications and are characterized by *sparse* constraint matrices with rational data. About half of the sample problems have no apparent special structure; the remainder show structural characteristics which are not exploited directly by our computational procedures. For all of our test problems we have computed a zero-one solution *and* proved it to be optimal with respect to the given linear objective function.

The test problems were solved on an IBM 370/168 at the Thomas J. Watson Research Center, Yorktown Heights, New York. All test problems were solved in less than one hour of CPU time; indeed, most problems were solved in less than ten CPU minutes. The experimental software package PIPX that we designed to solve pure zero-one programming problems uses the IBM linear programming software system MPSX/370 [6] and the branch-and-bound integer programming package MIP/370 [5] as "building blocks". (It goes without saying that *any comparable* commercial or non-commercial software packages for linear and integer programming can be used in lieu of MPSX/370 and MIP/370.)

The methodology that we have incorporated in PIPX includes automatic problem preprocessing and constraint generation. Problem preprocessing essentially consists of "inspecting" the user-supplied formulation of a zero-one linear program and improving upon the associated linear programming formulation by "tightening" the constraint set, by spotting variables which can be fixed at either zero or one, and by determining constraints of the problem which are rendered inactive by the previous data manipulations. Constraint generation essentially consists of generating cutting-planes which are satisfied by the zero-one solutions of the problem and which chop off part of the feasible set of the linear programming relaxation. The cutting planes that we use, however, are not the traditional cutting-planes found in textbooks on integer programming; rather, we have implemented recent theoretical results regarding the facial structure of zero-one polytopes; see, e.g., [9] for a related survey and detailed references. In addition, we have implemented a branch-and-bound strategy for MIP/370 which is designed to quickly find good integer solutions. This procedure is, in most cases, used repeatedly and utilizes information contained in the reduced costs associated with the optimal solution of the linear programming relaxation to fix variables to zero or one. The various procedures which comprise the PIPX computer system are performed automatically and no manual intervention is required.

The objective of this computational study is to establish the usefulness of the above mentioned methodological advances — when combined with clever branch-and-bound strategies — for the automatic solution of sparse large-scale zero-one linear programming problems and, most of all, to demonstrate that today's technology permits one to solve such problems, even when no apparent special structure is present, by *exact* methods in *economically feasible* solution times. The results of this study support this

hypothesis and are thus in line with the results of a similar study on the symmetric travelling salesman problem [1] where we made substantial use of the special structure of the problem and arrived at the same conclusion for this class of particularly hard combinatorial optimization problem.

The zero-one programming problems that we consider here have the following form:

(P) minimize $\{cx \mid Ax \leq b, x_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n\}$

where $A$ is an $m$-by-$n$ matrix with arbitrary rational entries, and $b$ and $c$ are vectors of length $m$ and $n$, respectively, with rational entries. In all test problems, the cost $c_j$ are nonnegative numbers. The problems that we are interested in have a *sparse* constraint matrix $A$, i.e., the total number of nonzero elements $a_{ij}$ of $A$ divided by the product of $m$ and $n$ is typically less than 0.05. Furthermore, while $A$ is permitted to have rows with entries equal to $+1$, $-1$ and 0 only — such as, for example, given by special ordered set constraints — the problems that we address here primarily have a substantial number of rows for which the nonzero elements of $A$ have a significant variance within each row. Indeed, if $A$ has only $+1$, $-1$ and 0 entries, or if all but a few rows of $A$ have this property, then neither problem preprocessing nor constraint generation as discussed in this study can be expected to be very effective. This is indeed the case for several of our test problems having this property. However, as far as constraint generation is concerned, the recent theoretical results regarding the facial structure of the polytopes associated with such problems, or with zero-one problems having a *dense* constraint matrix, suggest quite different methods than the ones that we have incorporated in our software package PIPX. The numerical results of our study show that for such problems, improvements in the branch-and-bound procedure can offer substantial computational gains.

In Section 2 we discuss the details of the problem preprocessing routine and indicate some promising avenues for the extension of what we have incorporated in PIPX. In Section 3 we describe the constraint generation procedure that we use and state some interesting theoretical problems which are suggested by the computational work and which, to date, are not yet resolved rigorously. In Section 4 we describe the special features of the branch-and-bound strategy that we used for MIP/370 and how we use the reduced costs of the linear programming relaxation to speed up the branch-and-bound phase. In Section 5 we describe how we integrated the various components to form the PIPX experimental computer system for the solution of pure zero-one programming problems. In Section 6 we describe our computational experiments in detail. Finally, in Section 7, we draw some general conclusions from our computational work.

# 2 Problem Preprocessing

Many researchers have pointed to the desirability of automatically making *ad hoc* improvements to formulations of integer programming problems prior to using linear programming-based solution methods. To obtain improved formulations, various possible strategies can be pursued ranging from complicated search procedures to rather straightforward but nevertheless computationally effective methods. In our computer program for the solution of pure zero-one programming problems, we have incorporated only the more obvious possibilities which we will describe briefly below. The fact that our uncluttered but effective procedure produced favorable results indicates that the topic of problem preprocessing deserves further attention.

## 2.1 Constraint Classification

When the data of problem (P), described in Section 1, are read into the computer, we first classify the inequalities into two types: Type 1 constraints are *special ordered set constraints*, i.e., constraints of the type

$$\sum_{j \in L} x_j - \sum_{j \in H} x_j \leq 1 - |H| \tag{2.1}$$

where $L$ and $H$ are disjoint index sets and $|H|$ denotes the cardinality of the set $H$. Clearly $x_j = 1$ for some $j \in L$ implies $x_k = 0$ for all $k \in L$, $k \neq j$, and $x_k = 1$ for all $k \in H$; while $x_j = 0$ for some $j \in H$ implies $x_k = 1$ for all $k \in H$, $k \neq j$, and $x_k = 0$ for all $k \in L$. In the present form of our computer program we do not require that two distinct special ordered set constraints are disjoint. Type 2 constraints are all other constraints of problem (P).

## 2.2 Variable Fixing and Blatant Infeasibility Check

Suppose that we have a type 2 constraint of problem (P) and write it for notational simplicity as

$$\sum_{j \in N_+} a_j x_j + \sum_{j \in N_-} a_j x_j \leq a_0 \tag{2.2}$$

where we have denoted by $N_+$ the index set of coefficients $a_j$ with positive value and $N_-$ the index set of coefficients $a_j$ with negative value. Clearly, if

$$\sum_{j \in N_-} a_j > a_0 \tag{2.3}$$

holds, then (2.2) does not have a feasible solution and the overall problem (P) of which (2.2) is but one constraint is *blatantly infeasible*. On the other hand, if

$$\sum_{j \in N_+} a_j \leq a_0 \tag{2.4}$$

holds, then the constraint (2.2) is *inactive* because every possible zero-one vector $x$ satisfies (2.2). Such an inequality can be dropped from the constraint set of (P) because it does not exclude any zero-one solution. Let $j \in N_+$ and suppose that

$$a_j > a_0 - \sum_{k \in N_-} a_k \tag{2.5}$$

holds; then $x_j = 0$ in every feasible zero-one solution to (2.2) and we can fix variable $x_j$ at the value zero and drop it from problem (P). Likewise, if for some $j \in N_-$ we have

$$-a_j > a_0 - \sum_{k \in N_-} a_k \tag{2.6}$$

then $x_j = 1$ holds in every feasible zero-one solution to (2.2). We can fix variable $x_j$ at the value one, adjust the right-hand-side vector $b$ of (P) and drop the variable $x_j$ from problem (P). If a variable that is fixed at value 1 appears also in a type 1 constraint with a positive coefficient, the remaining variables in this special ordered set are fixed as discussed in Section 2.1; a similar argument holds if a variable that is fixed at value 0 appears also in a type 1 constraint with a negative coefficient. All type 2 constraints of problem (P) are examined one at a time in the order they appear in the formulation.

## 2.3 Coefficient Reduction

Consider an arbitrary linear inequality in the form

$$\sum_{j=1}^{p} a_j x_j \geq a_0 \tag{2.7}$$

where all $a_j$ for $j = 1, \ldots, p$ are *positive*. If for some $k \in \{1, \ldots, p\}$ we have $a_k > a_0$ we can replace $a_k$ by $a_0$ and the inequality

$$a_0 x_k + \sum_{\substack{j=1 \\ j \neq k}}^{p} a_j x_j \geq a_0 \tag{2.8}$$

has the same solution set in terms of zero-one solutions as (2.7), but fewer real solutions in the unit-hypercube. Thus (2.8) is a "tighter" inequality than (2.7) for the associated linear programming relaxation. Of course, the constraints of (P) are not always of the form (2.7), but using the substitution $x_j = 1 - x_j$ where necessary, we can bring every constraint of (P) into the form (2.7), apply the reasoning above and check each coefficient of each type 2 constraint for a possible coefficient reduction. Then the transformation is reversed and the changed coefficients are recorded. The details of this procedure as they apply to problem (P) are easy to derive and are omitted here. The computer implementation is straightforward and the procedure has proved extremely efficient on real-world problems where the user employed the various "Big M" formulation devices which can be found — typically without any discussion of the impact of the choice of magnitude of M upon computation — in standard texts on integer programming.

## 2.4 Further Possible Enhancements

Clearly, the coefficient reduction described above can be improved by taking into account special ordered set constraints. While such an extension is not difficult to implement, it is not incorporated in our present software system. The fixing of variables described in Section 2.2 can be extended to detect, for example, constraints of the form

$$x_i + x_k \leq 1 \tag{2.9}$$

which can be extremely useful in limiting and guiding the search for zero-one solutions in the branch-and-bound phase. These inequalities are called *degree-two constraints*; see, e.g., the survey article [10]. In [8] systems of inequalities in zero-one variables with exactly two non-zero coefficients per inequality are studied from a theoretical point of view and strong *clique-inequalities* are derived which can be used in computation. Furthermore, there is no reason to limit oneself to the derivation of such implications from single inequalities only; see, e.g., [3] where implications of the form (2.9) are derived from all pairs of the original constraints. It is, however, clear that any additional work which is done to *preprocess* a given problem of the form (**P**) must be weighed against the possible benefits that are obtained in terms of an improved linear programming formulation of the problem and against the possible benefits that are obtained in terms of guiding and limiting the search for a zero-one solution in the branch-and-bound phase of a solution procedure. While an implementation of the latter requires more flexible branch-and-bound codes than exist today — at least as regards commercially available codes for branch-and-bound — some of these possible benefits can be reaped as well through the generation of cutting planes which we discuss in the next section.

# 3 Constraint Generation

Considerable research activity in the past ten years or so has focused on the facial structure of zero-one polytopes; see, e.g., [9] for a related survey article and further references. Briefly, the related studies are concerned with valid inequalities for various zero-one linear programming problems which are *best possible* ones in the sense that they are required in any representation of the convexified solution set of the problem under consideration by way of linear inequalities. While a theoretically satisfactory *and* computationally useful characterization of *all* linear inequalities which describe the convexified solution set of the constraints of problem (P) will probably prove to be elusive, partial results in that direction have been obtained to date and, as the experimental results of this paper demonstrate, these partial results do an extremely useful job in the numerical solution of problems with a sparse matrix $A$ and no apparent special structure.

## 3.1 Sparse Zero-One Problems and Knapsack Problems

The zero-one problem (P) with a single linear constraint, i.e., the case where $m=1$, is called the *knapsack problem* and the facial structure of the associated polytope has been thoroughly studied, even though a complete list of the linear inequalities which define the knapsack polytope is still unknown to date. Denote by $(a^i,b_i)$ the $i$-th row of $(A,b)$ and assume for simplicity that all constraints of (P) are type 2 constraints. We define by

$$P_I^i = \text{conv}\{x \in \mathcal{R}^n \mid a^i x \leq b_i,\ x_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n\} \tag{3.1}$$

the convex hull (i.e., the convexified solution set; short: conv ) of the zero-one solutions to the single inequality $a^i x \leq b_i$ where $i \in \{1, \ldots, m\}$. $P_I^i$ is the knapsack polytope associated with constraint $i$ of problem (P). Likewise, we denote by

$$P_I = \text{conv}\{x \in \mathcal{R}^n \mid Ax \leq b,\ x_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n\} \tag{3.2}$$

the convex hull of zero-one solutions to the entire constraint set of problem (P). $P_I$ is the zero-one polytope associated with problem (P) and clearly we have

$$P_I \subseteq \bigcap_{i=1}^{m} P_I^i \tag{3.3}$$

i.e., $P_I$ is contained in the intersection of all of the knapsack polytopes $P_I^i$, $i = 1, \ldots, m$. Equality in (3.3) does *not*, in general, hold, but *does* hold if, for example, problem (P) decomposes totally into $m$ knapsack problems. This is the case if every variable $x_j$ appears in exactly one of the $m$ constraints $a^i x \leq b_i$ for $i = 1, \ldots, m$. If we have a large-scale zero-one programming problem with a *sparse* matrix $A$ and with *no apparent special structure*, it is reasonable to expect that the intersection of the $m$ knapsack polytopes $P_I^i$ provides a fairly good approximation to the zero-one polytope $P_I$ over which we wish to minimize a linear objective function. This is our working hypothesis and our computational results confirm that it is a reasonable assumption. Practically speaking, this assumption permits us to concentrate

on the individual rows of the constraint set of problem (P) when we derive valid inequalities for the polytope $P_I$. While the proceeding is clearly *valid* for any zero-one problem (P), it cannot be expected to lead to substantial computational gains if, for instance, the matrix $A$ is rather dense or if almost all entries in $A$ equal $+1$, $-1$ and 0. In these two cases the constraint generation must be done differently; see [9]. For, in the latter case, e.g., if *all* components of a row $a^i$ of $A$ equal $+1$, $-1$ and 0, then we know that

$$P_I^i = \{x \in \mathbb{R}^n \mid a^i x \leq b_i,\ x_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n\} \tag{3.4}$$

holds provided that $b_i$ is integer, and no further improvement in terms of approximating $P_I$ by way of linear inequalities can be obtained from using such a row *individually*. On the other hand, if the matrix $A$ is *dense* then the different rows of $A$ interact and cutting planes from individual rows of $A$, while certainly valid and in some instances useful, cannot be expected to produce the same impressive results that we get for sparse large-scale zero-one problems with no apparent special structure.

## 3.2 Facets of the Knapsack Polytope

We consider a single inequality $a^i x \leq b_i$ of the constraint system of problem (P) and, using the variable substitution $x_j = 1 - x_j$ where necessary, we bring the inequality into a form where all nonzero coefficients are positive. Dropping the index $i$ for notational convenience we consider thus a linear inequality

$$\sum_{j \in K} a_j x_j \leq a_0 \tag{3.5}$$

where the $a_j$ are positive rational numbers and the variables $x_j$ assume the values zero or one. Let $S \subseteq K$ be such that

$$\sum_{j \in S} a_j > a_0 \text{ and } \sum_{j \in S} a_j - a_k \leq a_0, \text{ for all } k \in S \tag{3.6}$$

hold. Then $S$ is called a *minimal cover* with respect to (3.5) and obviously every zero-one solution to (3.5) satisfies the inequality

$$\sum_{j \in S} x_j \leq |S| - 1 \tag{3.7}$$

where $|S|$ denotes the cardinality of the set $S$. Suppose next that $S^* \subseteq K$ and $t \in K - S^*$ satisfy

$$\sum_{j \in S^*} a_j \leq a_0 \textit{ and } Q \cup \{t\} \textit{ is a minimal cover for every } Q \subseteq S^* \textit{ with } |Q| = k \tag{3.8}$$

where $k$ is an integer number satisfying $2 \leq k \leq |S^*|$. Due to the one-element role of the index $t$ and because $k$ is some integer number, the set $S^* \cup \{t\}$ is called a *(1,k)-configuration* with respect to (3.5). Every zero-one solution to (3.5) satisfies the following set of inequalities which are associated with a (1,k)-configuration:

$$(r - k + 1)x_t + \sum_{j \in T(r)} x_j \leq r \tag{3.9}$$

where $T(r) \subseteq S^*$ varies over all subsets of cardinality $r$ of $S^*$ and $r$ varies over all integers from $k$ to $|S^*|$, inclusively. If $k = |S^*|$ holds in (3.8), then a (1,k)-configuration is a minimal cover; thus, in general, the class of inequalities associated with (1,k)-configurations contain properly the class of inequalities associated with minimal covers. Both inequalities (3.7) and (3.9) are *best possible* ones if $K=S$ holds in the first case, or if $K = S^* \cup \{t\}$ holds in the second case, i.e., in these cases the respective inequalities define *facets* of the associated knapsack polytope; see [9]. In general, however, these inequalities must be "lifted" to obtain facets of the knapsack polytope associated with (3.5), i.e., they must be extended appropriately to the variables $x_j$ with index $j$ in $K-S$, or with index $j$ in $K-S^*-\{t\}$, respectively.

The extension is done over the respective variables by the following recursive *lifting procedure:* Initially we set $f_j = 1$ for all $j \in S$, $f_0 = |S|-1$ *or* $f_j = 1$ for all $j \in S^*$, $f_t = (r-k+1)$, $f_0 = r$ and $S = S^* \cup \{t\}$, respectively. For the iterative step let $k \in K-S$ and determine

$$z_k = \max\{\sum_{j \in S} f_j x_j \mid \sum_{j \in S} a_j x_j \leq a_0 - a_k,\ x_j = 0 \text{ or } 1 \text{ for all } j \in S\} \tag{3.10}$$

Define $f_k = f_0 - z_k$, redefine $S$ to be $S \cup \{k\}$ and repeat until $K-S$ is empty. The inequality $fx \leq f_0$ that results defines a facet for the polytope $P_I^i$ associated with (3.5). Obviously, the solution of (3.10) requires the solution of several zero-one problems. In the next section we discuss how to identify suitable minimal cover inequalities and return later to the question of how to efficiently approximate the lifting procedure.

We note that the *support* of an inequality obtained by lifting (3.7) or (3.9) is contained in the support of the inequality (3.5), i.e., $f_j \neq 0$ holds only if $a_j \neq 0$ holds. Thus the inequalities that we generate preserve the sparsity of the constraint matrix. This is an additional difference to the traditional cutting planes described in the textbooks on integer programming. The latter are typically rather dense and — as integer programming folklore has it — lead to explosive storage requirements. The results in Section 6 show that the constraints which we generate lead to moderate increases in storage requirements.

## 3.3 Constraint Identification

To use the methods described above in actual computation we could, of course, simply generate *a priori all* possible minimal covers and (1,k)-configurations for each row of the problem (P), extend the respective inequalities in some fashion as, e.g., discussed above, append them to the linear constraints of (P) and proceed to solve the *augmented* linear programming problem. Clearly this would be wasteful and, for large-scale problems, impossible to implement: The number of possible (1,k)-configurations for (3.5) is evidently exponential in the number of variables of the constraint (3.5) and thus storage requirements would simply explode. Thus there is no other possibility than to generate such constraints on the "fly", i.e. to generate them in the course of computation as they are needed. To this end we start by solving the linear program

$$\min\{cx \mid Ax \leq b,\ 0 \leq x_j \leq 1 \text{ for } j = 1, \ldots, n\} \tag{3.11}$$

and obtain an optimal solution $\bar{x}$ to (3.11). If $\bar{x}$ is a zero-one solution, we stop: $\bar{x}$ solves the problem (**P**). Else we take any type 2 row of $A$ and solve the following problem —

***Constraint Identification Problem:*** *Given $\bar{x}$ find a minimal cover inequality (3.7) or a (1,k)-configuration inequality (3.9) which chops off $\bar{x}$, if such an inequality exists.*

The constraint identification problem is solved in turn for *each* row of the original constraint matrix $A$ that qualifies, the resulting inequalities that are identified are "lifted" and appended to the linear programming problem, the augmented problem is reoptimized and the procedure is repeated until we find a zero-one solution, until no more constraint is found, or until the gain in the objective function value becomes too small, e.g., is less than one unit in terms of the units of the objective function. And, of course, when one of the latter two possibilities occur we resort to branch-and-bound.

The question is then to find a procedure which solves the constraint identification problem. Using the notation of Section 3.2, consider the zero-one knapsack problem

$$\min \{ \sum_{j \in K} (1-\bar{x}_j)\, s_j \mid \sum_{j \in K} a_j\, s_j > a_0,\ s_j = 0 \text{ or } 1 \text{ for all } j \in K \} \tag{3.12}$$

where the inequality is strict in the knapsack constraint. It follows that there exists a minimal cover inequality (3.7) which chops off $\bar{x}$ if and only if the optimal objective function value of (3.12) is less than one. For suppose there exists a minimal cover $S \subseteq K$ which chops off $\bar{x}$. Setting $s_j = 1$ for all $j \in S$, $s_j = 0$ for all $j \in K-S$ we find that (3.12) has an optimal objective function value less than one. To prove the reverse direction, observe that $0 \leq \bar{x}_j \leq 1$ for all $j = 1, \ldots, n$ implies that the objective function coefficients in (3.12) are nonnegative, and hence among the optimal solutions to (3.12) there exists at least one which defines a minimal cover for (3.5). Let $S$ be the set of variables with value 1 in such a solution. If the optimum value of the objective function value of (3.12) is less than 1 it follows that the corresponding inequality (3.7) chops off $\bar{x}$. As can easily be verified, problem (3.12) is constructed in such a manner that its solution finds a *most violated* minimum cover inequality.

Our formulation of the constraint identification problem for minimal cover inequalities involves the solution of a zero-one knapsack problem for which, to date, no technically good algorithm is known. We *conjecture* that — using a different approach than the one outlined here — violated minimal cover inequalities can be identified by a polynomially bounded algorithm. For (1,k)-configurations we have at present not even a formulation of the constraint identification problem in a tractable form such as (3.12). Rather we use at present an *ad hoc* procedure for finding (1,k)-configurations which can be improved upon.

## 3.4 Implementation of Constraint Identification

To implement the theoretical development outlined above we have written a Fortran program which does the constraint generation. Given the current solution $\bar{x}$ we scan each type 2 constraint and set up auxiliary data structures to minimize the actual computational effort. First, using the substitution $x_j = 1 - x_j$ where necessary, we bring the row under consideration into the form (3.5) with all nonnegative coefficients. This changes the solution vector from $\bar{x}$ to, say, $\bar{y}$. Next, define

$$K_1 = \{j \in N \mid \bar{y}_j > 0,\ a_{ij} \neq 0\},\ K_0 = \{j \in N \mid \bar{y}_j = 0,\ a_{ij} \neq 0\} \tag{3.13}$$

where $N = \{1, \ldots, n\}$ and $i$ is the index of the current type 2 constraint. Clearly to solve the constraint identification problem it is sufficient to work with the data in $K_1$ only which is a substantial advantage since $|K_1|$ is usually small for sparse problems. If $K_1$ is empty, if $j \in K_1$ implies $\bar{y}_j = 1$, or if $j \in K_1$ implies $|a_{ij}| = 1$, then there is nothing to do in row $i$ and we process the next type 2 constraint. Otherwise, we solve problem (3.12) with $K$ replaced by $K_1$ and $\bar{x}$ replaced by $\bar{y}$, respectively, as a *linear program,* i.e., we replace the constraints $s_j = 0$ or 1 by $0 \leq s_j \leq 1$ for all $j \in K_1$. This problem is a linear knapsack problem which is solved by Dantzig's method [2, p. 517]. Rounding the fractional variables to 1 we obtain a cover for the knapsack which we proceed to change into a minimal cover by dropping some of the variables to zero. We call $S$ the index set of the variables in the minimal cover.

Having obtained a minimal cover $S$ we proceed to sort the variables in $S$ by increasing order of magnitude of the coefficients $|a_{ij}|$, i.e., by decreasing ratio $1/|a_{ij}|$. This is done in order to have a start for the lifting procedure. If $K_1 - S$ is nonempty, we extend the minimal cover inequality (3.7) to the variables in $K_1 - S$ using the lifting procedure described in Section 3.2. As in the previous case, we relax the zero-one problem (3.10) to its associated linear program, i.e., we replace $x_j = 0$ or 1 by $0 \leq x_j \leq 1$ for all $j \in S$. Again, this yields a linear knapsack problem which we solve by Dantzig's method. Denote $\bar{z}_k$ the optimal objective function value of this linear program and $z_k^*$ the value obtained from $\bar{z}_k$ by truncating it to its integer part. Clearly $z_k$, the optimal objective function of (3.10), satisfies $z_k \leq z_k^*$. We define $f_k$ to be $f_0 - z_k^*$. Because $z_k \leq z_k^*$ holds, we *underestimate* the "true" extension coefficient and thus the extension is valid because of the nonnegativity of the constraint. The possible error that we make is small since the coefficients of minimal cover inequalities are all less than $|S|$ and, typically, $|S|$ is small. If the resulting extension coefficient $f_k$ is positive, we merge variable $k$ into the list of variables according to its ratio $f_k/|a_{ik}|$ and iterate until all variables in $K_1 - S$ are processed. If $f_k$ equals zero, the variable $k$ need no longer be considered in the lifting procedure. Note that the iterative step of the lifting procedure can be executed very quickly because the "next" step continues to use the ordered list of variables of the previous step in the Dantzig solution to the linear program associated with (3.10).

After having processed all variables in $K_1 - S$, or if $K_1 - S$ is empty, we check whether or not the resulting (extended) minimal covering inequality chops off $\bar{y}$. If this is not the case, we try to find a (1,k)-configuration; see below. Else we extend the inequality to the variables with index in $K_0$ using the lifting procedure in relaxed form as described previously. After having obtained a violated inequality, we store it and proceed with the next available type 2 constraint.

The identification of (1,k)-configurations in our present implementation is preliminary at best and goes as follows: We start with a minimal cover $S$ identified earlier and define the index $t$ of the (1,k)-configuration to be the index with the biggest $|a_{ij}|$ for all $j \in S$. If this does not yield a unique index we go to the next type 2 constraint. Otherwise, we scan the elements in the set $K_1$ and attempt to enlarge the minimal cover to a set $S^*$ such that (3.8) is satisfied with respect to the index $t$ that we have chosen. If this is not possible, we process the next type 2 constraint. Otherwise, we check inequality (3.9) to see whether or not it chops off the solution $\bar{y}$. If not, we continue to enlarge $S^*$. When this is no longer possible we pick the next type 2 constraint. If a violated (1,k)-configuration was found, we extend the corresponding inequality to the other variables in $K_1 - S^* - \{t\}$ using the lifting procedure described above. We then extend the inequality to the variables in $K_0$ and store the resulting constraint. We next try to enlarge $S^*$ further in order to find more violated (1,k)-configurations in the current row and proceed as described above until the process stops. Then the next type 2 constraint is processed.

The order of computations thus involves always as a first step the identification of a minimal cover. If successful, we take the next type 2 constraint. If not successful, we look for a (1,k)-configuration. And so on. At the end of the constraint generation phase we have thus a list of violated inequalities from minimal covers and/or (1,k)-configurations from the individual rows of the problem (P). These new constraints are then added to the constraint set of the linear programming problem. The constraints which are generated in the course of this computation are not used for further constraint generation; they are simply appended to the linear program and not further classified.

Special ordered set constraints (type 1) are not fully exploited in our current implementation. At present we use them in the following fashion: When a type 2 constraint is set up for constraint generation, we check whether or not the "first" variable with a nonzero coefficient is contained in some special ordered set (SOS), where "first" is relative to the numbering order used. If so, we check whether or not the corresponding SOS is satisfied at equality by the current solution $\bar{x}$. If not or if the variable is not contained an any SOS, we check the next variable in the numbering order. Otherwise we determine the smallest coefficient in the overlap of the SOS and the current constraint having a positive $\bar{x}$ component, take its absolute value and add the SOS multiplied by this number to the current constraint. We flag all variables of the current type 2 constraint which are contained in this SOS constraint as having been processed, and continue the process with the next unflagged variable of the *original* type 2 constraint. It is possible that we create new nonzeros in the constraint this way and this is taken into consideration. In other words, we form a *surrogate constraint* from the original type 2 constraint and a non-overlapping subset of SOS constraints which are tight at the current solution. This surrogate constraint is then used for constraint generation in lieu of the original one. Evidently one can do better and in a recent paper [7] we have given a generalization of the Dantzig method to linear knapsack problems with SOS constraints. Also, we have studied in [7] some properties of the associated zero-one polytope. These results were not available when we began to work on this study and thus have not been incorporated in our implementation.

# 4 Branch-and-Bound Methods

We describe now how we modified the standard branch-and-bound algorithm of MIP/370 and applied it to the augmented zero-one problem described in Section 3.3. In most cases, the augmented problem will have the desired characteristic that the gap between the continuous linear program optimal objective function value and the optimal zero-one objective function value is relatively small, thereby making the branch-and-bound search for an optimal integer solution easier. In order to further facilitate the search, the following modifications and enhancements are made to the standard MIP/370 algorithm:

- The computation of an *ad hoc* upper bound on the optimal zero-one solution.
- The PIP Strategy for quickly finding integer solutions.
- The use of continuous reduced cost implications to fix variables.

## 4.1 The Ad Hoc Upper Bound

As we have at present no reasonable heuristic for general large-scale zero-one linear programs which could be used to determine a true upper bound for the optimal zero-one solution, we compute an *ad hoc* bound for the zero-one optimum from the objective function value of the augmented linear programming relaxation. This bound is assigned to the MIP/370 variable XMXDROP, which is the value of the objective function beyond which a branch or node of the branch-and-bound tree is dropped forever; see [5]. The value for the computed *ad hoc* bound is

$$\text{XMXDROP} = \text{XPCTGAP} * \text{XFUNCT} \tag{4.1}$$

where XFUNCT is the value of the optimal continuous linear program objective function and XPCTGAP is a parameter with value in the range 1.1 to 1.25.

The assumption underlying this rather simple bounding mechanism is that the gap between the augmented continuous linear program optimal objective function value and the optimal zero-one objective function value is small. Furthermore, if, for a particular problem, the optimal zero-one solution is not within ten to twenty-five per cent of the continuous solution, then presumably the preprocessing and constraint generation procedures have not been effective for the problem at hand, and we have no desire to exhaust the branch-and-bound tree by MIP/370 processing for a truly large-scale problem. Should indeed our assumption be wrong, then we can at least expect to complete the run for the problem at hand with the additional useful information that the optimal zero-one objective function value is greater than our computed bound.

As indicated in the Computational Experiments in Section 6, for those problems for which the preprocessing and constraint generation procedures were effective, the afore mentioned objective function gap always fell within the above limit and (4.1) proved to be a valid bounding mechanism. Indeed, in all of our test problems, it sufficed to choose XPCTGAP equal to 1.1.

## 4.2 The PIP Strategy

The PIP branch-and-bound strategy is a way of guiding MIP/370 to quickly find integer solutions. The primary assumption of the PIP strategy is that, as previously mentioned, the gap between the optimal continuous linear program solution value and the optimal zero-one solution value is relatively small, i.e., that the *augmented* linear program provides a fairly good approximation to the zero-one polytope in the neighborhood of an optimal zero-one solution.

To understand the PIP strategy, we must introduce some MIP/370 terminology. (This discussion is applicable to minimization problems.) The following MIP/370 variables, all of which contain floating-point numeric values, are related, as indicated, to how MIP/370 searches for integer solutions:

XMXDROP — As indicated previously, any node or branch whose objective function value exceeds XMXDROP is abandoned forever in the search.

XMXBESTF — At any time during the search, XMXBESTF contains the best (minimal) functional value of the waiting nodes. This implies that the best integer solution that can still be expected cannot have a functional value less than XMXBESTF.

XMXSCAN — A node in the branch-and-bound tree is *postponed* (it will not be considered for branching) if its objective function value is greater than XMXBESTF+XMXSCAN. Note: a node which is postponed as a branching candidate because of this rule may later become eligible if XMXSCAN or XMXBESTF is increased.

The relationship between XMXDROP, XMXBESTF and XMXSCAN is illustrated in Figure 1.

Areas A, B and C in Figure 1 show where the objective function values of nodes in the branch-and-bound tree may be with respect to XMXDROP, XMXBESTF and XMXSCAN. Nodes in area A are eligible candidates for branching; nodes in area B are postponed and are not candidates for branching; area C contains no nodes because nodes and branches with objective function values greater than XMXDROP are dropped from the tree.

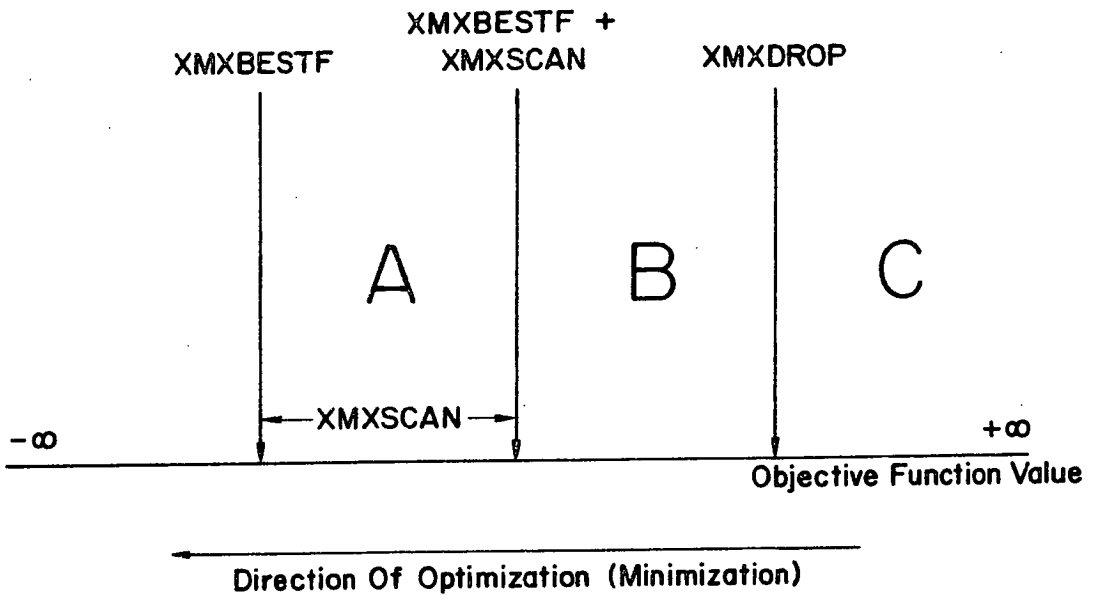


**Figure 1: Relationship Between MIP/370 Variables**

The PIP strategy starts with

$$\text{XMXSCAN} = 0.1 * (\text{XMXDROP} - \text{XMXBESTF}) \qquad (4.2)$$

In other words, area A in Figure 1 is small compared to area B, resulting in nodes with very good objective function values being the branch node candidates. XMXSCAN is gradually increased until every waiting node is a candidate for branching. That is, area A in Figure 1 includes all nodes between XMXBESTF and XMXDROP; there is no area B. At the same time that XMXSCAN is increasing, a higher priority is given to nodes based on their integer infeasibility; see [5] as amended by IBM Technical Newsletter SN19-1132, April 30, 1979, p. 146.2. After a predetermined number of nodes have been developed and if no integer solution has been found, XMXSCAN is reset to the value indicated in (4.2) and the process is repeated.

## 4.3 Continuous Reduced Cost Implications

In a pure zero-one minimization linear programming problem, given the continuous optimal objective function $Z_{lp}$, the continuous optimal reduced costs $d_j$, and a true upper bound on the zero-one optimal objective function value $Z^+$, the following implications hold for variables $x_j$ which are nonbasic in the continuous optimal solution:

1. If $x_j = 0$ in the continuous solution and $Z^+ - Z_{lp} < d_j$ then $x_j = 0$ in every optimal zero-one solution.

2. If $x_j = 1$ in the continuous solution and $Z^+ - Z_{lp} < -d_j$ then $x_j = 1$ in every optimal zero-one solution.

In the branch-and-bound routine, the test above is made each time a zero-one solution is obtained and $Z^+$ is set equal to the objective function value of the zero-one solution that was found. If more than a fixed percentage, typically ten per cent, of the current active variables can be fixed, the problem is revised by setting the indicated variables to zero or one, and the procedure is restarted by performing preprocessing and constraint generation on the new problem. In this case the previously generated additional cutting planes are purged from the computer memory, i.e., we restart from scratch on the *reduced original* problem. We do this because it is easier to implement and because the generation of constraints can be done rather quickly — the generation of cutting planes definitely did not create a bottleneck in our computations.

It would seem, however, that it is much more efficient to be able to fix the indicated variables in the current branch-and-bound tree and continue the search, rather than restarting from the beginning with the new problem after revision. Unfortunately, there is no way to perform such a global function in MIP/370. The fact that we are willing to abandon a search tree and restart the entire procedure with a reduced problem reflects our philosophy that we would rather do almost *anything* than perform branch and bound. Our philosophy is not a prejudice; is is based on our experience solving this type of problem. The excellent computational times reported in Section 6 confirm our strategy.

# 5 The PIPX Computer System

PIPX is an experimental computer software system, written in PLI, which calls MPSX-MIP/370 procedures and Fortran routines to perform the preprocessing, constraint generation and branch-and-bound functions described in Sections 2, 3 and 4. It is another example of the successful use of the algorithmic and data handling features of a state-of-the-art, commercial linear programming package as building blocks for larger application systems; see, e.g., [1,4].

PIPX consists of 22 external procedures (6 written in PLI, 5 in Fortran, and 11 being MPSX-MIP/370 procedures). The name, purpose, calling procedure and procedures called for the non-MPSX-MIP/370 procedures in PIPX are summarized in Table 1. For a more detailed description of MPSX-MIP/370 procedures, see [5,6].

PIPX requires two types of data storage: MPSX/370 data storage, used by the computational routines of MPSX-MIP/370, and PIPX internal data storage. The MPSX/370 data storage component is allocated automatically by the SETUP routine and consists mainly of work regions and buffers; see [6]. PIPX internal data storage is allocated dynamically, based on problem size and storage availability, and includes two sparse representations of the linear programming problem (row-major order and column-major order schemes) and utility storage used for the problem preprocessing and constraint generation functions.

Figure 2 illustrates the overall program control and data flow for PIPX. A more detailed functional description of the system, with emphasis on the program control and data manipulation interface to MPSX/370, will appear in a subsequent publication. The following is a brief functional overview of PIPX and serves as an annotation of Figure 2.

1. The zero-one linear programming problem, in SHARE format [6, p. 193], is read from file by CONVERTX. The appropriate internal data structures, including row-major order and column-major order representations of the problem, are allocated and initialized.

2. The PROBLEM PREPROCESSOR uses the internal data structures to perform the functions described in detail in Section 2.

3. Decision T1 determines if the problem has been found infeasible in the PROBLEM PREPROCESSOR. If so, PIPX terminates.

4. REDUCE1 updates the internal representation of the problem, eliminating variables which have been fixed and constraints which have become inactive due to the actions of the PROBLEM PREPROCESSOR.

5. CONVERT, an MPSX/370 procedure, uses the internal representation of the problem to create the MPSX/370 representation on PROBFILE (the problem file used by MPSX/370).

| Name | Purpose | Called by | Calls to procedures in | |
|---|---|---|---|---|
| | | | PIPX | MPSX-MIP/370 |
| OPTIPIP | main procedure | — | PIPIN<br>CORDER<br>ROWFOR<br>DETSOS<br>REDCOF<br>PIPRED<br>PIPRV<br>PIPCVT<br>PIPCGN | TIME<br>CONVERT<br>SOLUTION<br>SELIST<br>MIXSTART<br>MIXFLOW<br>MIXSTATS |
| PIPIN | reads the problem in SHARE format from MPSX/370 file SYSIN and sets up intermediate sparse data structures | OPTIPIP | — | — |
| CORDER | called after PIPIN to set up the sparse column representation of the linear programming tableau | OPTIPIP<br>PIPRED | — | — |
| ROWFOR | called after PIPIN to set up the sparse row representation of the linear programming tableau | OPTIPIP<br>PIPRED | — | — |
| DETSOS | SOS constraint determination and infeasibility checking | OPTIPIP | — | — |
| REDCOF | coefficient reduction | OPTIPIP | — | — |
| PIPRED | elimination of inactive rows and fixed variables prior to calling MPSX/370 procedure REVISE | OPTIPIP | CORDER<br>ROWFOR | — |
| PIPRV | computes intra-row variance for active, non-SOS constraints | OPTIPIP | — | — |
| PIPCVT | builds a PLI structure interface to MPSX/370 routine CONVERT | OPTIPIP | — | — |
| PIPCGN | iteratively solves linear programming problems augmented with constraints | OPTIPIP | CONGEN | SETUP<br>RESTORE<br>OPTIMIZE<br>SOLUTION<br>REVISE |
| CONGEN | constraint generation | PIPCGN | — | — |

**Table 1: PIPX External Procedures**

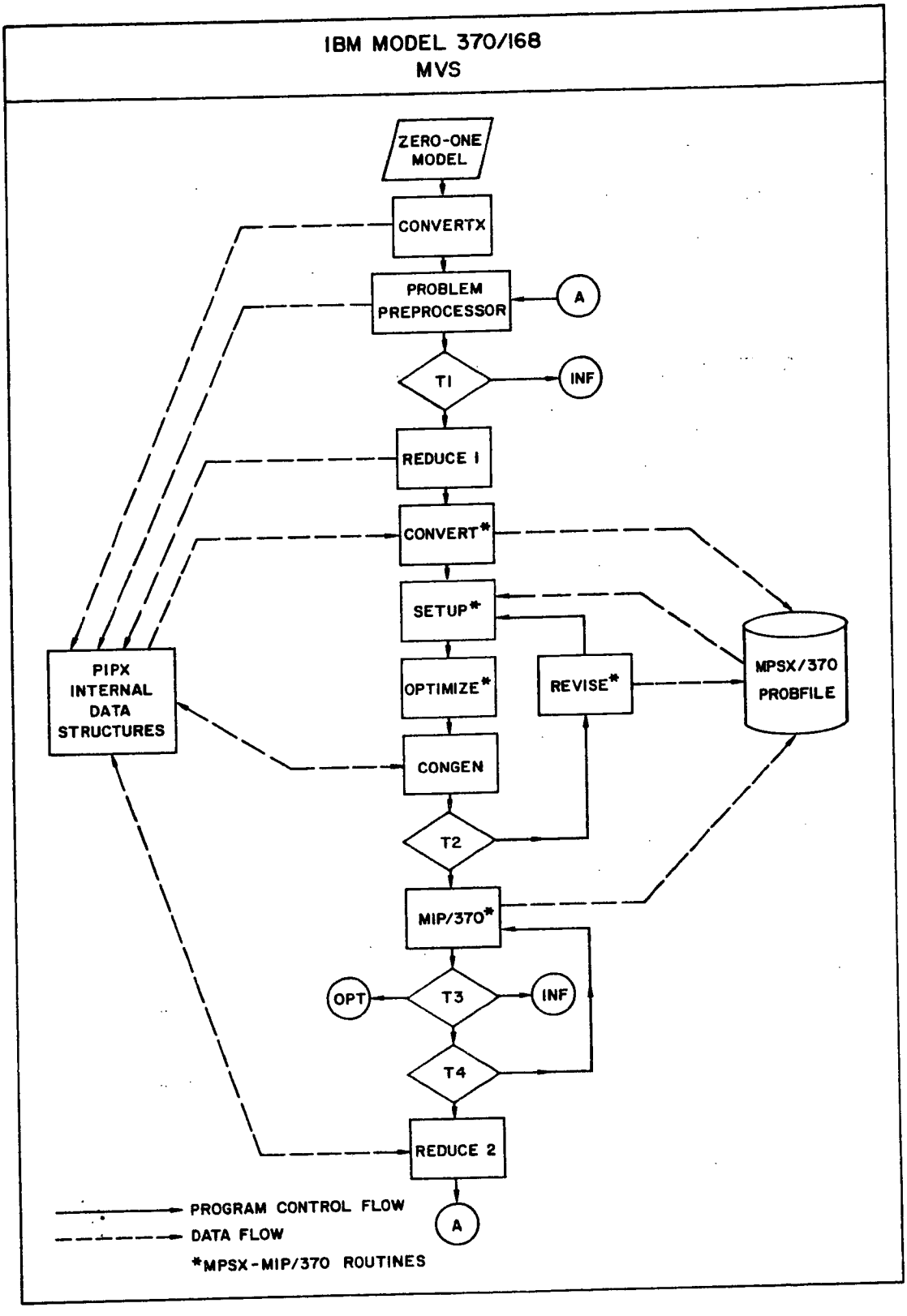


**Figure 2: PIPX Control and Data Flow**

6. SETUP, an MPSX/370 procedure, brings the problem in from PROBFILE and creates the MPSX/370 in-storage data structures.

7. OPTIMIZE, an MPSX/370 procedure, solves the continuous linear programming problem.

8. The problem and the optimal continuous solution are then processed by CONGEN in the manner described in Section 3.4. The actions determined by decision T2 are either

   i) update the problem with the new constraints, using the MPSX/370 procedure REVISE, and restart the sequence from the SETUP step, or

   ii) determine that no constraints have been generated, or that the constraints generated do not have a significant effect of the linear programming objective function value; in either case, the constraint generation phase is terminated.

   **Note:** In most cases, the SETUP, OPTIMIZE, CONGEN, REVISE sequence is repeated several times, resulting in successive linear programming problems with increasingly greater (in the case of minimization) objective function values; see Section 6.

9. MIP/370 is invoked to solve the augmented zero-one linear programming problem; details of the strategy employed are given in Section 4.

10. Decision T3 is invoked when MIP/370 terminates or when an integer solution is found. The possible actions are

    i) PIPX terminates with an optimal solution, or

    ii) PIPX terminates because the problem is zero-one infeasible, or

    iii) a nonoptimal integer solution is found, in which case decision T4 is invoked. Decision T4 determines if the reduced costs from the continuous optimal solution can be used to fix variables and thus reduce the problem; see Section 4.3. If such reduction is indicated, the problem is processed by REDUCE2, and the PIPX program control sequence is restarted from the PROBLEM PREPROCESSOR step. Otherwise, MIP/370 branch-and-bound processing continues.

# 6 Computational Experiments

The PIPX computer system was applied to a set of real large-scale zero-one linear programming problems. Table 2 summarizes the main characteristics of the test problem set. Under the heading VARS, ROWS and NONSOS appear the number of variables, the total number of constraints, and the number of those constraints which are *not* of the special ordered set type. Under the heading VARIANCE we state the *average* standard deviation of the absolute values of the nonzero elements of the non-SOS rows of the constraint matrix after preprocessing; the standard deviation of this aggregate statistic appears in parenthesis. Thus, for example, P2756 has 756 rows out of which 386 are non-SOS rows. Computing for each non-SOS row of P2756 the standard deviation of the absolute values of its nonzero elements and averaging over all 386 rows yields an average intra-row standard deviation of 149.2, i.e., a substantial variation of the nonzero coefficients; the number 89.4 indicates that *among* the non-SOS rows to the constraint matrix the variation is substantial, but somewhat less than on average *within* each non-SOS row. In problems P1550, P1939 and P2655 the respective figures indicate that among the respective two non-SOS rows there is only one which contributes to the standard deviation, the other row having identical nonzero elements. Indeed, these problems are set partitioning problems with two additional constraints; one of which constrains the sum of all variables to be a certain number, the other one being a knapsack constraint with positive coefficients which, however, have little variation. Under the heading DENS appears the density of the original constraint matrix plus the cost row; i.e., the total number of nonzero elements of $A$ and $c$ divided by the product of $m$ and $n$ and multiplied by 100. Finally, $Z_{lp}$ denotes the optimal objective function value of the linear programming relaxation of problem (P) in original form, i.e., prior to applying any of the PIPX functions, while $Z_{ip}$ denotes the optimal zero-one objective function value.

| NAME | VARS | ROWS | NON-SOS | VARIANCE* | DENS | $Z_{lp}$ | $Z_{ip}$ |
|---|---|---|---|---|---|---|---|
| P0033 | 33 | 16 | 11 | 49.6 (33.9) | 17.4 | 2520.6 | 3089.0 |
| P0040 | 40 | 24 | 13 | 39.6 (92.8) | 11.3 | 61796.5 | 62027.0 |
| P0201 | 201 | 134 | 107 | 0.7 (1.0) | 5.0 | 6875.0 | 7615.0 |
| P0282 | 282 | 242 | 44 | 34.3 (16.2) | 2.0 | 176867.5 | 258411.0 |
| P0291 | 291 | 253 | 14 | 3.5 (2.3) | 1.9 | 1705.1 | 5223.75 |
| P0548 | 548 | 177 | 94 | 123.9 (88.3) | 1.8 | 315.3 | 8691.0 |
| P1550 | 1550 | 94 | 2 | 8.4 (8.4) | 7.4 | 1706.5 | 1708.0 |
| P1939 | 1939 | 109 | 2 | 3.5 (3.5) | 4.8 | 2051.1 | 2066.0 |
| P2655 | 2655 | 147 | 2 | 2.3 (2.3) | 3.4 | 6532.1 | 6548.0 |
| P2756 | 2756 | 756 | 386 | 149.2 (89.4) | 0.4 | 2688.7 | 3124.0 |

*After problem preprocessing

**Table 2: Test Problem Summary**

The computational experiments described here were executed on the IBM 370/168 computer, running the MVS operating system, at the Thomas J. Watson Research Center in Yorktown Heights, New York. Fortran programs were compiled using the Fortran H Extended Compiler. PLI programs were compiled using the PLI Optimizing Compiler.

## 6.1 Zero-One Preprocessing Phase

The zero-one preprocessor is the initial computational phase in the PIPX system. As indicated in Section 2, the effect of the preprocessor is to tighten the user-supplied linear programming formulation of the zero-one problem, to eliminate inactive constraints, and to fix variables at either zero or one. Table 3 summarizes the results of the initial preprocessor step on the test problem set. Under the heading "ORIGINAL PROBLEM" are the number of rows and variables in the problem before preprocessing. Under the heading "ELIMINATED" are the number of rows and variables which were successfully eliminated from the problem. The time for the initial preprocessor phase, in CPU minutes, is given under "TIME".

| NAME | ORIGINAL PROBLEM | | ELIMINATED | | TIME* |
|---|---|---|---|---|---|
| | ROWS | VARS | ROWS | VARS | |
| P0033 | 16 | 33 | 0 | 0 | 0.01 |
| P0040 | 24 | 40 | 0 | 0 | 0.02 |
| P0201 | 134 | 201 | 0 | 6 | 0.01 |
| P0282 | 242 | 282 | 20 | 0 | 0.02 |
| P0291 | 253 | 291 | 47 | 1 | 0.02 |
| P0548 | 177 | 548 | 20 | 21 | 0.03 |
| P1550 | 94 | 1550 | 0 | 0 | 0.19 |
| P1939 | 109 | 1939 | 0 | 0 | 0.18 |
| P2655 | 147 | 2655 | 0 | 0 | 0.30 |
| P2756 | 756 | 2756 | 17 | 22 | 0.50 |

*CPU minutes, IBM 370/168 - MVS

**Table 3: Problem Preprocessor Summary**

In several cases, the zero-one preprocessor was reapplied to a revised problem after variables were eliminated using the continuous reduced cost implications (see Section 4.3). We have omitted the detailed results of any second and subsequent applications of the preprocessor procedure because such data are not crucial to this discussion. Table 7 in Section 6.4, however, gives the number of major passes of the PIPX system for each test problem, and hence the number of times the zero-one preprocessor was invoked.

## 6.2 Constraint Generation Phase

The constraint generation procedure is the second computational phase of the PIPX system; it operates on the preprocessed problem to produce and solve a linear programming problem with a better (greater in the minimization case) optimal continuous objective function value (see Section 3). Table 4 summarizes the results of the first constraint generation step on the test problem set. Under "PREPROCESSED PROBLEM" are the number of constraints, the number of variables, the density of the problem, and optimal continuous objective function value $Z^{*}_{lp}$ of the problem produced by the preprocessor, before constraint generation. Under "AUGMENTED PROBLEM" are the number of constraint generation "passes", i.e., the number of intermediate linear programming problems required, the number of constraints in the final linear programming problem produced, the density of its constraint matrix including the cost row, and its optimal continuous objective function value $Z^{**}_{lp}$. The time for the initial constraint generation phase, in CPU minutes, is given under the heading "TIME". This time includes the solution and reoptimization of all linear programming calculations in the constraint generation phase, including the original linear program.

| NAME | PREPROCESSED PROBLEM | | | | AUGMENTED PROB | | | | TIME* |
|---|---|---|---|---|---|---|---|---|---|
| | VARS | ROWS | DENS | $Z^{*}_{lp}$ | PASS | ROWS | DENS | $Z^{**}_{lp}$ | |
| P0033 | 33 | 16 | 17.4 | 2819.4 | 6 | 36 | 12.1 | 3065.3 | 0.12 |
| P0040 | 40 | 24 | 11.3 | 61829.1 | 4 | 29 | 13.8 | 61862.8 | 0.12 |
| P0201 | 195 | 134 | 5.0 | 7125.0 | 1 | 139 | 5.0 | 7125.0 | 0.16 |
| P0282 | 282 | 222 | 1.5 | 176867.5 | 14 | 462 | 1.3 | 255033.1 | 1.00 |
| P0291 | 290 | 206 | 1.3 | 1749.9 | 6 | 278 | 2.5 | 5022.7 | 0.33 |
| P0548 | 527 | 157 | 1.9 | 3125.9 | 9 | 296 | 1.3 | 8643.5 | 0.57 |
| P1550 | 1550 | 94 | 7.4 | 1706.5 | 1 | 94 | 7.4 | 1706.5 | 0.81 |
| P1939 | 1939 | 109 | 4.8 | 2051.1 | 1 | 110 | 4.8 | 2051.1 | 0.74 |
| P2655 | 2655 | 147 | 3.4 | 6532.1 | 2 | 149 | 3.4 | 6535.0 | 2.04 |
| P2756 | 2734 | 739 | 0.4 | 2701.1 | 8 | 1065 | 0.4 | 3115.3 | 3.06 |

*CPU minutes, IBM 370/168 - MVS

**Table 4: Constraint Generation Summary**

As in the case of the preprocessing phase, fixing variables using the continuous reduced cost implications in the branch-and-bound phase causes the constraint generation procedure to be reapplied to the revised problem. As before, the details of second and subsequent applications of the constraint generation procedure are omitted.

The real measure of the effectiveness of the constraint generation procedure is determined by how much it closes the "gap" between the optimal linear programming relaxation objective function value and the optimal zero-one objective function value. These data are summarized in Table 5 for the first application of the constraint generation procedure. Under the heading "ΔROWS" are the number of constraints generated by the procedure. The "GAP" is the difference between the optimal zero-one objective function value and the optimal continuous objective function value before constraint generation but after problem preprocessing, i.e., the GAP is $Z_{ip}-Z^{*}_{lp}$. The figure "RATIO" is computed as follows:

$$\text{RATIO} = (Z^{**}_{lp}-Z^{*}_{lp})/(Z_{ip}-Z^{*}_{lp})$$

Thus RATIO measures the fraction of the GAP that was closed due to the generation of cutting planes. For example, the application of constraint generation to problem P2756 produced 326 new constraints and closed 98 per cent of the gap.

| NAME | ΔROWS | GAP | RATIO |
|---|---|---|---|
| P0033 | 20 | 269.6 | 0.92 |
| P0040 | 5 | 197.9 | 0.17 |
| P0201 | 5 | 490.0 | 0.00 |
| P0282 | 240 | 81543.5 | 0.96 |
| P0291 | 72 | 3473.8 | 0.94 |
| P0548 | 139 | 5565.1 | 0.99 |
| P1550 | 0 | 1.5 | 0.00 |
| P1939 | 1 | 14.9 | 0.00 |
| P2655 | 2 | 15.9 | 0.19 |
| P2756 | 326 | 422.9 | 0.98 |

**Table 5: Effect of Constraint Generation**

## 6.3 Branch-and-Bound Phase

The application of MIP/370 to perform branch-and-bound is the third computational phase in the PIPX system. Because of the necessity to revise a problem after variables are fixed using continuous reduced cost implications, most of the test problems required more than one application of the branch-and-bound procedure. Table 6 summarizes the branch-and-bound phase and the related continuous reduced cost implication results. Under "BRANCH-AND-BOUND" are the total number of zero-one solutions obtained (both optimal and nonoptimal) and the total number of branch-and-bound tree nodes produced for all applications of the branch-and-bound procedure. Under the heading "DJ FIX" are the statistics for the continuous reduced cost implication procedure. Under "PASSES" is the total number of times the continuous reduced costs were used to reduce the problem. Under "ΔROW" and "ΔVAR" are the total number of constraints and variables eliminated due to continuous reduced cost implications. Since the problem is subsequently processed again by the preprocessor, additional variables are fixed and additional constraints are dropped. These statistics are included in Table 6.

| NAME | BRANCH-AND-BOUND | | DJ FIX | | |
|---|---|---|---|---|---|
| | SOLUTIONS | NODES | PASSES | ΔROW | ΔVAR |
| P0033 | 2 | 113 | 0 | 0 | 0 |
| P0040 | 1 | 11 | 1 | 2 | 15 |
| P0201 | 2 | 1116 | 0 | 0 | 0 |
| P0282 | 4 | 1862 | 0 | 0 | 0 |
| P0291 | 1 | 87 | 1 | 164 | 127 |
| P0548 | 1 | 36 | 1 | 66 | 261 |
| P1550 | 2 | 10 | 2 | 0 | 1442 |
| P1939 | 2 | 334 | 2 | 0 | 1746 |
| P2655 | 5 | 214 | 2 | 0 | 1967 |
| P2756 | 3 | 2392 | 1 | 260 | 1341 |

**Table 6: Branch-and-Bound Summary**

## 6.4 PIPX Execution Summary

Table 7 summarizes the time, in CPU minutes, for PIPX system execution on the test problem set. Under "PIPX Passes" is the number of major iterations performed by PIPX. (See Section 5) Under "TIME" are the aggregate times spent in each of the three main PIPX computational phases, and the total time required to solve the test problems.

| NAME | PIPX Passes | TIME* | | | |
|---|---|---|---|---|---|
| | | Pre-processor | LP/Constraint Generation | Branch-and-Bound | Total |
| P0033 | 1 | 0.01 | 0.12 | 0.56 | 0.69 |
| P0040 | 2 | 0.03 | 0.13 | 0.02 | 0.18 |
| P0201 | 1 | 0.01 | 0.16 | 9.90 | 10.07 |
| P0282 | 1 | 0.02 | 1.00 | 11.70 | 12.72 |
| P0291 | 2 | 0.04 | 0.51 | 0.30 | 0.85 |
| P0548 | 2 | 0.12 | 0.70 | 0.09 | 0.91 |
| P1550 | 3 | 0.50 | 0.90 | 0.10 | 1.50 |
| P1939 | 3 | 0.49 | 0.86 | 13.64 | 14.99 |
| P2655 | 3 | 0.82 | 3.23 | 2.70 | 6.75 |
| P2756 | 2 | 1.07 | 4.03 | 49.32 | 54.42 |

*CPU minutes, IBM 370/168 - MVS

**Table 7: PIPX Execution Summary**

# 7 Conclusions

All test problems that we considered during the course of this study were solved to optimality. The methodology that we propose for the solution of large-scale zero-one linear programming problems produced — by today's standards — impressive computational results, in particular on sparse problems having no apparent special structure. The test problems were given to us from various sources within and outside of the IBM Corporation and, while we have no comparative data on previous solution attempts, most were originally considered not amenable to exact solution in economically feasible computation times. The computational results reported in Section 6 contradict this sentiment and strongly confirm our hypothesis that a combination of problem preprocessing, cutting planes and clever branch-and-bound techniques permit the optimization of sparse large-scale zero-one linear programming problems, even when no apparent special structure is present, in reasonable computation times. Our results indicate that cutting-planes are an indispensable tool for the exact solution of this class of problem. A case in point is the problem P0548 where the preprocessing closed a considerable portion of the gap between the continuous objective function value of the user-supplied linear programming formulation and the integer objective function value. Undoubtedly, however, preprocessing alone would not have permitted us to optimize P0548 in about one minute of CPU time. The remaining gap is still much too large to permit one to expect completion of the branch-and-bound phase within a reasonable time limit. Indeed, we originally attempted to solve this problem without the use of cutting planes and had to give up after letting the problem run for several CPU hours. After the identification of cutting planes, the gap after preprocessing was reduced to a small fraction of the original value; this, combined with branch-and-bound improvements, permitted us to solve the problem — quite unexpectedly — in less than one minute of CPU time. This observation is not a singular occurrence; it is supported by the results of several other successful runs including problem P2756 having 2756 zero-one variables.

Inspection of the problem characteristics given in Table 2 reveals that the cutting planes considered here work best when we have both a low density *and* a substantial variance among the nonzero coefficients of the constraint matrix. For the latter we use the statistics given by the preprocessed version of the problem and *not* the user-supplied version, because the cutting planes are generated from the preprocessed problem. Whenever these criteria are not met the constraint generation as implemented in our experimental software system PIPX did not have a substantial impact upon our computations. The reason why in these cases we did get the impressive computation times reported here lies in the various improvements that we implemented over the standard MIP/370 algorithm, including the use of the reduced cost implications for limiting the search; see the discussion in Section 4. However, problem P0033 is a notable exception to the behavior of problems with such properties, indicating that even for dense problems one can *sometimes* expect substantial improvements of the linear programming relaxation by the use of cutting planes derived from the individual rows of the constraint matrix. As mentioned in Section 3, it would be wrong to conclude that cutting planes cannot be used to aid in the solution of problems having a dense matrix or an an insignificant variance among the nonzero coefficients of the

constraint matrix; rather, for this type problem, the constraint generation must be carried out differently from the way we currently generate cutting planes.

In the course of this study, we encountered two major implementational difficulties. The first was the constraint identification procedure discussed in Section 3.3. While the method used here to identify violated constraints is an *ad hoc* procedure, it appears to be very effective in actual computation; it is easy to implement and we do not hesitate to advocate its use for the solution of sparse large-scale problems displaying a substantial intra-row variance of the nonzero coefficients. It is clear, however, that special ordered set (SOS) constraints, when present, should be taken into consideration explicitly for the constraint generation as well as for the preprocessing of zero-one problems. Our experience with the use of such constraints indicates that it is preferable to work with a *flexible* SOS structure rather than with a *fixed* or *user-supplied* SOS structure, in particular as regards the constraint identification. If the identification problem discussed in Section 3 can be solved rigorously by a polynomially bounded algorithm, this should go a long way towards making the exact solution of sparse large-scale pure zero-one linear programming problems a routine task.

The second major difficulty was the design and implementation of an effective and efficient interface between our computational procedures and MPSX-MIP/370. The experimental software system PIPX which we implemented was concerned not only with performing the mathematical procedures described previously, but also with the management of storage and control flow in concert with MPSX-MIP/370. This task was facilitated by the use of the Extended Control Language (ECL) of MPSX/370 (ECL is, in fact, a superset of PLI). For simple applications, the design and implementation of ECL programs for utilizing MPSX-MIP/370 functions is usually straightforward. For an application on the scale of PIPX, however, the task is more complex and requires a certain amount of "experimentation" to arrive at an effective and efficient system. The fact that we were able to realize such a system demonstrates anew the feasibility of using commercial linear programming package components as building blocks in specialized applications.

Finally, we are confident that the exact solution method discussed in this paper permits one to solve substantially larger zero-one linear programming problems than the ones we attacked in this study, in economically feasible computation times. We limited ourselves to the solution of real-world models because we seriously doubt the validity of testing such methods on randomly generated problems.

# References


[1] Crowder, H. and M.W. Padberg, "Solving Large-Scale Symmetric Travelling Salesman Problems to Optimality", *Management Science 26* (1980), 495-509.

[2] Dantzig, G.B., *Linear Programming and Extensions,* Princeton University Press (1963).

[3] Hammer, P.L., Padberg, M.W. and U.N. Peled, "Constraint Pairing in Integer Programming", *INFOR - Can. J. of Operations Res 13* (1975) 68-81.

[4] Ho, J.K. and E. Loute, "An Advanced Implementation of the Dantzig-Wolfe Decomposition Algorithm for Linear Programming", *Mathematical Programming 20* (1981) 303-326.

[5] *IBM Mixed Integer Programming/370 (MIP/370) Program Reference Manual,* Form number SH19-1099 (IBM Corporation, 1975).

[6] *IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual,* Form number SH19-1095 (IBM Corporation, 1979).

[7] Johnson, E.L. and M.W. Padberg, "A Note on the Knapsack Problem with Special Ordered Sets", to appear in *Operations Research Letters* (1981).

[8] Johnson, E.L. and M.W. Padberg, "Degree-Two Inequalities, Clique Facets and Biperfect Graphs", Report No. 80167, Universität Bonn, FRG (December 1980).

[9] Padberg, M.W., "Covering, Packing and Knapsack Problems", *Annals of Discrete Mathematics 4* (1979) 265-287.

[10] Spielberg, K., "Enumerative Methods in Integer Programming", *Annals of Discrete Mathematics 5* (1979) 139-183.